SIMULATIONS AMONG MULTIDIMENSIONAL ITERATIVE ARRAYS
ITERATIVE TREE AUTOMA (U) ILLINOIS UNIV AT URBANA
COORDINATED SCIENCE LAB J L TRAHAN JAN 86
UNCLASSIFIED UILU-ENG-86-2282
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A
SIMULATIONS AMONG MULTI-DIMENSIONAL ITERATIVE ARRAYS, ITERATIVE TREE AUTOMATA, AND ALTERNATING TURING MACHINES

Jerry Lee Trahan
**Report Title:** Simulations among multidimensional iterative arrays, iterative

1. **Personal Author(s):** Trahan, Jerry L.
2. **Type of Report:** Technical
3. **Time Covered:** 1986 January
4. **Page Count:** 42

**Abstract:**
We present three simulations: a simulation of an alternating Turing machine (ATM) operating in time $T(n)$ by an iterative tree automaton (ITA) in time $O(T(n))$, a simulation of a d-dimensional iterative array (dIA) operating in time $T(n)$ by an ATM in time $O(T(n))$, and a simulation of an ITA operating in time $T(n)$ by an ATM in time $O(T(n))$. The first two improve previously known results. The first implies the simulation of a nondeterministic Turing machine by an ITA in time $O(T(n))$ of Culik and Yu (1984). The second is stronger than the simulation of a dIA by an ATM in time $O(T(n))/logT(n)$ of Seiferas (1977) and Dymond and Tompa (1985).
11. tree automata, and alternating turing machines
SIMULATIONS AMONG MULTIDIMENSIONAL ITERATIVE ARRAYS.
ITERATIVE TREE AUTOMATA.
AND ALTERNATING TURING MACHINES

BY
JERRY LEE TRAHAN
B.S., Louisiana State University and A. & M. College. 1983

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign. 1986

Accession For

Urbana. Illinois
We present three simulations: a simulation of an alternating Turing machine (ATM) operating in time $T(n)$ by an iterative tree automaton (ITA) in time $O(T(n))$, a simulation of a $d$-dimensional iterative array (dIA) operating in time $T(n)$ by an ATM in time $O((T(n))^d)$, and a simulation of an ITA operating in time $T(n)$ by an ATM in time $O((T(n))^2)$. The first two improve previously known results. The first implies the simulation of a nondeterministic Turing machine by an ITA in time $O(T(n))$ of Culik and Yu (1984). The second is stronger than the simulation of a dIA by an ATM in time $O((T(n))^{d+1}/\log T(n))$ of Seiferas (1977) and Dymond and Tompa (1985).
ACKNOWLEDGEMENTS

I wish to thank my thesis advisor, Dr. Michael C. Loui, for his encouragement, guidance, and constructive suggestions for this thesis. I also wish to thank my parents for their constant support throughout my education. This work was supported in part by the Office of Naval Research under contract N00014-85-K-0570.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. DEFINITIONS</td>
<td>3</td>
</tr>
<tr>
<td>3. LITERATURE REVIEW</td>
<td>7</td>
</tr>
<tr>
<td>4. THE ITA SIMULATION OF THE DIA</td>
<td>9</td>
</tr>
<tr>
<td>5. THE ATM SIMULATION OF THE ITA</td>
<td>28</td>
</tr>
<tr>
<td>6. CONCLUSIONS AND OPEN PROBLEMS</td>
<td>34</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>36</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Page

Figure 4.1 - Table of state transitions from state $q_r$ ............................................ 17
Figure 4.2 - Table of state transitions from state $q_u$ ............................................ 18
Figure 4.3 - Computation array $C$ for 1-dimensional case ........................................ 20
Figure 4.4 - $\theta D$, input to procedure ASIMD

for 1-dimensional case .................................................................................. 22

Figure 4.5 - Values known in computation array $D$ after Step 3 of ASIMD ............. 24
Figure 5.1 - Contents of ATM tapes in procedure ASIMT ....................................... 30
Multidimensional iterative arrays, iterative tree automata, and alternating Turing machines are important models of parallel computation. A *d-dimensional iterative array* (dIA) comprises an infinite set of finite state machines located at the points of the d-dimensional integer lattice. An *iterative tree automaton* (ITA) consists of an infinite set of finite state machines connected into a binary tree. An *alternating Turing machine* (ATM), like a *nondeterministic Turing machine* (NTM), may have choices of transitions for each combination of state, input symbol, and worktape symbols. From an *existential* state an ATM accepts if at least one choice leads to an accepting state; from a *universal* state an ATM accepts if every choice leads to an accepting state. One can view the ATM as a computational model that makes copies of itself to evaluate each of the choices. A *deterministic Turing machine* (DTM) has only a single possible transition for each combination of state, input symbol, and worktape symbols. Each of these models has a fixed structure. Each processor (finite state machine or ATM copy) can communicate with only a fixed set of other processors, in contrast to variable structure models such as the hardware modification machines of Dymond and Cook (1980) and the parallel random access machines of Fortune and Wyllie (1978).

This thesis studies the simulation of a dIA by an ITA. The simulation of an ATM by an ITA and the simulation of a dIA by an ATM achieve this purpose. Let $X(t)$ denote the set of all languages recognized by machines of type $X$ in time $O(t)$, where $X \in \{\text{dIA, ITA, ATM, DTM, NTM}\}$. These simulations produce the following time bounds:

$$\text{ATM}(t) \subseteq \text{ITA}(t) \text{ and }$$

$$\text{dIA}(t) \subseteq \text{ATM}(t^d).$$

In addition, this thesis presents the simulation of an ITA by an ATM within the bound
ITA(t) \subseteq ATM(t^2), and considers the simulation of an ATM by a dIA.

Two of the above time bounds improve previously known results. The first, ATM(t) \subseteq ITA(t), implies the bound NTM(t) \subseteq ITA(t) established by Culik and Yu (1984) because an NTM is more restricted than an ATM. The second, dIA(t) \subseteq ATM(t^d), is better than dIA(t) \subseteq ATM(t^{d+1}/\log t). The latter bound arises from the combination of a result of Seiferas (1977a), who proved that dIA(t) \subseteq DTM(t^{d+1}), and a result of Dymond and Tompa (1985), who established that DTM(t) \subseteq ATM(t/\log t).

Aside from improving upon previous findings, the results described in this thesis are significant because they extend current knowledge about ATMs. In addition to the usual computational resources time and space (which generally are inversely related), the ATM has alternations between universal and existential states as a computational resource. The ATM simulation of the dIA and the ATM simulation of the ITA utilize the resource alternation to achieve the stated time bounds.

The ITA simulation of an ATM uses the ITA’s capability to emulate direct central control, that is, to act as though the state transitions of each finite state machine in the ITA depend on the state of a uniquely designated finite state machine. The techniques of Seiferas (1977b) are used to establish the direct central control capability of the ITA.

The ATM simulation of the dIA uses the “divide-and-conquer” technique, as in Savitch (1970), Paterson (1972), and Loui (1981).

The framework of this thesis is as follows. Chapter 2 gives definitions of the dIA, the ITA, and the ATM. Chapter 3 reviews the literature related to the thesis research. Chapters 4 and 5 detail the simulations themselves, prove their correctness, and determine their running times. Chapter 6 describes the conclusions reached from this research and offers some open research problems.
Chapter 2
DEFINITIONS

To more precisely describe the actions of the computational models in the simulations to follow, this chapter defines the dIA, the ITA, and the ATM.

Let $n$ be the length of the input string. For all $n$, a machine recognizes an input string of length $n$ in time $T(n)$ if the machine requires at most $T(n)$ steps to accept the input string.

A $d$-dimensional iterative array (dIA) is an infinite synchronized $d$-dimensional array of finite state machines, one for each $d$-tuple of nonnegative integers. In dIA $M$ let $M(X)$ denote the machine corresponding to the $d$-tuple $X$, called the coordinates of $M(X)$. Let $O$ denote the $d$-tuple of all 0s. Only $M(O)$ receives input symbols and produces output symbols. At each step $M(O)$ reads a new symbol of the input string or does not read a new symbol as a function of the current state of itself, the current states of its neighbors, and the current input symbol. A neighbor of a machine with coordinates $X$ in the array is a machine whose coordinates are obtained by adding or subtracting 1 from a single coordinate of $X$. The state transitions of every machine depend on the current states of the machine itself and its $2^d$ neighbors. The state transitions of $M(O)$ depend additionally on the current input symbol. A dIA accepts an input string in $\Sigma^*$ when $M(O)$ enters a state in a designated set of final states.

Formally, a dIA $M$ is a septuple $M = (d, Q, \Sigma, \delta, \rho, r, F)$ where

- $d$ is a positive integer.
- $Q$ is a finite set of states.
- $\Sigma$ is a finite input alphabet ($\$ \in \Sigma$ is an endmarker, $B \in \Sigma$ is the blank symbol).
- $\delta_o: \Sigma^{2^d-1} \times (\Sigma \cup \{B, \$\}) \to Q$ is the transition function for $M(O)$, the machine at the origin.
- $\delta: \Sigma^{2^d-1} \to Q$ is the transition function for every machine other than $M(O)$.
- $r: \Sigma^{2^d-1} \times (\Sigma \cup \{B, \$\}) \to \{true, false\}$ is the function that specifies whether $M(O)$ reads the
next input symbol, and

\[ F \subseteq Q \] is a set of final states.

\[ q_\lambda \in Q \] is a special quiescent state. Initially, every machine is in a quiescent state. \( \delta \) satisfies the condition \( \delta(q_\lambda, q_{\lambda_1}, \ldots, q_{\lambda_n}) = q_\lambda \) thus except for \( M(Q) \), a machine leaves the quiescent state only after a neighbor leaves the quiescent state. Infinite B's are assumed to follow the end of the input string.

An iterative tree automaton (ITA) is composed of synchronized finite state machines connected in an infinite full binary tree structure. Let \( R(\lambda) \) denote the root of ITA \( R \), where \( \lambda \) is the empty string. In general, for any finite binary string \( \beta \) let \( R(\beta 0) \) and \( R(\beta 1) \) denote the left and right children of \( R(\beta) \), respectively. The level of \( R(\beta) \) is \( |\beta| \), the length of \( \beta \). Define \( |\lambda| = 0 \). Only \( R(\lambda) \) receives input symbols and produces output symbols. At each step \( R(\lambda) \) reads a new symbol of the input string or does not read a new symbol as a function of the current state of the root, the current states of its two children, and the current input symbol. The current state of the root, the current states of its two children, and the current input symbol determine the next state of \( R(\lambda) \). For every other machine in the ITA, the current states of the machine itself, its parent and its left and right children determine the next state of the machine. An ITA accepts an input string when \( R(\lambda) \) enters any one of a designated set of final states.

Formally, an ITA \( R \) is a septuple \( R = (Q, \Sigma, \delta_0, \delta, \delta, r, F) \) where

\( Q \) is a finite set of states,

\( \Sigma \) is a finite input alphabet (\( $ \in \Sigma \) is an endmarker, \( B \in \Sigma \) is the blank symbol),

\( \delta_0 : Q \times (\Sigma \cup \{B, S\}) \times Q \rightarrow Q \) is the transition function for \( R(\lambda) \), the machine at the root,

\( \delta : Q \rightarrow Q^2 \) are the transition functions of the left and right children, respectively, of each machine \( R(\beta) \), \( \beta \in \{0, 1\}^* \),

\( r : Q \times (\Sigma \cup \{B, S\}) \times Q^2 \rightarrow \{true, false\} \) is the function that specifies whether the root reads the next input symbol, and
\( F \subseteq Q \) is a set of final states.

If \( X, W, Y, Z \) are respectively the current states of \( R(\beta 0) \), its parent \( R(\beta) \), its left child \( R(\beta 00) \), and its right child \( R(\beta 01) \), then \( \delta_t(X,W,Y,Z) \) is the next state of \( R(\beta 0) \). Similarly \( \delta_t \) specifies the next state of each \( R(\beta 1) \).

\( q_x \in Q \) is a special quiescent state. Initially, every machine in the ITA is in a quiescent state. \( \delta \) satisfies the condition \( \delta(q_x q_x q_x q_x q_x) = q_x \) where \( \delta \) is \( \delta_t \) or \( \delta_i \), thus except for \( R(\lambda) \), a machine leaves the quiescent state only after its parent leaves the quiescent state. Infinite \( B \)'s are assumed to follow the input string. The transition function for a left child must be different from the transition function for a right child for the ITA to be distinct from a 1TA.

A configuration of an ITA \( R \) is a pair \((C, w)\), where \( C \) is a mapping from the machines in \( R \) to \( Q \) and \( w \in \Sigma^* \). For all \( \beta \in \{0,1\}^* \) and \( X \in \{0,1\}^* \), where \( w \) is the unread portion of the input string, \( R \) has a legal transition from \((C, w)\) to \((C', w')\) if

1. \( C'(\lambda) = \delta_0(C(\lambda), a, C(0), C(1)) \).
2. \( C'(\beta 0) = \delta_t(C(\beta 0), C(\beta 00), C(\beta 01), C(\beta)) \).
3. \( C'(\beta 1) = \delta_i(C(\beta 1), C(\beta 10), C(\beta 11), C(\beta)) \), and
4. \( w = w' = a = \$ \), or
   \[ w = aw' \text{ if } r(C(\lambda), C(0), C(1), a) = \text{true} \] or
   \[ w = w' \text{ if } r(C(\lambda), C(0), C(1), a) = \text{false} \] for every \( (C, w) \to (C', w') \) if \( C(\beta) \neq C(\beta) \) for \( \beta \geq k \) since we start with all machines in a quiescent state.

A computation by ITA \( R \) is a sequence \((C_0, w_0), (C_1, w_1), (C_2, w_2), \ldots \) of configurations where the transition from \((C_i, w_i)\) to \((C_{i+1}, w_{i+1})\) is legal for all \( k \). Note that \( C(\beta) = C(\beta) \) for \( \beta \geq k \) since we start with all machines in a quiescent state.

An alternating Turing machine (ATM) is defined in Chandra, Kozen, and Stockmeyer (1981). A configuration of an ATM is the state of the ATM, the contents of the input tape, the contents of each of the worktapes, and the locations of each of the tape heads. An ATM is a Turing machine in which every nonfinal state is either universal or existential. A configuration with an existential state is accepting if at least one successor configuration is
accepting. A configuration with a universal state is accepting if every successor configuration is accepting. An ATM accepts an input string if its initial configuration is accepting. An ATM has a two-way read-only input tape with endmarkers and k worktapes, which are initially blank. A step of an ATM consists of reading one symbol from each worktape and reading an input symbol, then writing a symbol on each of the worktapes, moving each of the heads left or right one tape square or not moving the tape heads, and choosing a new state from the set specified by the transition function.

One can describe all possible computations of an ATM on some input string as a computation tree. All nodes are configurations, the root is the initial configuration, and the children of any configuration c are exactly those configurations that can be reached from c in one step according to the transition rules of the ATM. The leaves of the tree are the final configurations and may be accepting or rejecting. A branch of the computation tree is a downward directed path from the root; in other words, a branch is a sequence of configurations starting with the initial configuration. Assume that for an ATM to run in time T(n), all branches terminate in at most T(n) steps.

Formally, an ATM AM is a septuple $AM = (k, Q, \Sigma, \Gamma, \delta, q_0, g)$ where

- $k$ is the number of worktapes.
- $Q$ is a finite set of states.
- $\Sigma$ is a finite input alphabet ($\$ \in \Sigma$ is an endmarker).
- $\Gamma$ is a finite worktape alphabet ($B \in \Gamma$ is the blank symbol).
- $\delta : Q \times \Gamma^k \times (\Sigma \cup \{\$\}) \rightarrow P(Q \times (\Gamma \setminus \{ \text{blank} \})^k \times \{ \text{left}, \text{right}, \text{stationary} \}^k \times \{+1\})$ is the transition function, where $P(S)$ is the power set of $S$, that is, the collection of subsets of $S$.
- $q_0 \in Q$ is the initial state, and
- $g : Q \rightarrow \{ \text{universal}, \text{existential}, \text{accept}, \text{reject} \}$ is a mapping identifying each state as a universal, existential, accepting, or rejecting state.
Chapter 3

LITERATURE REVIEW

This chapter reviews the literature related to the research reported in this thesis.

Chandra, Kozen, and Stockmeyer (1981) present the concept of alternation. (The same authors originally presented the concept in Chandra and Stockmeyer (1976) and Kozen (1976).) This thesis uses their ATM model. They derive significant relationships between classes of languages accepted by time and space bounded ATMs and those accepted by time and space bounded DTMs. In particular, logarithmic alternating space is equivalent to polynomial deterministic time, and polynomial alternating time is equivalent to polynomial deterministic space. Paul, Prauss, and Reischuk (1980) demonstrate that an ATM with a single tape can simulate an ATM with multiple tapes in linear time.

Dymond and Tompa (1985) prove another result related to this thesis. They establish that $\text{DTM}(t) \subseteq \text{ATM}(t/\log t)$. Their proof associates the computation of the DTM with an acyclic directed graph. They use a two-person pebbling game to pebble the graph within a time bound of $O(n/\log n)$ for a graph with $n$ vertices. Next, the ATM steps simulate the two-person pebbling of the graph. In the pebbling game, one person's moves correspond to existential choices of the ATM, and the other person's moves correspond to universal choices of the ATM.

Paterson (1972) represents a TM computation as a two-dimensional diagram of successive tape configurations. He employs divide-and-conquer in both time and space dimensions. This method is generalized in this thesis in the simulation of a dIA by an ATM. Loui (1981) establishes a space bound for a DTM to accept the same language as a d-dimensional NTM with one worktape head. The proof utilizes a generalization of crossing sequences across the boundaries of d-dimensional boxes of the worktape. He uses a divide-and-conquer method to recursively partition the boxes.
Rosenfeld (1979) presents a good review of iterative automata.

Cole (1969) formally presents the d-dimensional iterative array of finite state machines. He establishes that the computing speed of a dIA can be increased by a constant factor by enlarging the set of states of each machine. He proves that the class of context-free languages does not contain all the sets of strings accepted by a dIA, nor do the sets of strings accepted by a dIA contain all context-free languages. He proves that computing capability increases as the number of dimensions increases.

Seiferas (1977a) extends Cole's work on deterministic dIAs to nondeterministic dIAs (NdIAs). He derives that

\[ \text{NTM}(t^d) \subseteq \text{NdIA}(t), \]

\[ \text{NdIA}(t) \subseteq \text{NTM}(t^{d-1}), \] and

\[ \text{dIA}(t) \subseteq \text{DTM}(t^{d-1}). \]

The second result is related to the simulation of an NdIA by an ATM given in this thesis. His simulation uses about \( n^d \) steps of a one-dimensional \((2d+1)\) head TM to simulate the \( n \)th step in a computation of a dIA.

Seiferas (1977b) establishes that a dIA with direct central control is no more powerful than a regular dIA, and that a regular dIA can simulate a dIA with direct central control in linear time. In this simulation, the finite state machine at the origin of the dIA controls the dIA indirectly by propagating the value of its state outward using only local communication.

Culik and Yu (1984) construct a language \( L \) such that an ITA accepts \( L \) in real-time, but no dIA can accept \( L \) in real-time. They state that the converse problem, that is, whether real-time dIA languages \((d \geq 2)\) are properly contained in real-time ITA languages, is an open problem. They establish that an ITA can simulate an NTM in linear time. Their simulation provides a basis for the simulation of an ATM by an ITA in Chapter 4.
Chapter 4

THE ITA SIMULATION OF THE DIA

This chapter contains a simulation of the ATM by the ITA and a simulation of the dIA by the ATM and proofs of the correctness of each simulation.

Lemma 1: Every language recognized in time \( t \) by a \( k \)-tape ATM can be recognized in time \( O(t) \) by a one-tape ATM.

This result is from Paul, Prauss, and Reischuk (1980). They specify that the one-tape ATM does not have separate input and output tapes.

Lemma 2: Every \( t \) steps of an ATM with at most \( c \geq 3 \) choices at each step can be simulated by \((c-1)t\) steps of an ATM with at most 2 choices at each step.

Proof: Let \( M = (k, Q, \Sigma, \Gamma, \delta, q_0, g) \) be an ATM with at most \( c \) choices at each step. We define an ATM \( M' = (k', Q', \Sigma, \Gamma, \delta', q_0', g) \) with at most 2 choices at each step such that \( M' \) simulates \( M \). Let a transition rule of \( M \) be

\[
\delta(q_0, w, a, n) = \{(q_1, w_1, X_1), (q_2, w_2, X_2), \ldots, (q_j, w_j, X_j)\}
\]

for \( q_0, q_1, \ldots, q_j \in Q, w, w_1, \ldots, w_j \in \Gamma^* \), \( a, a_0 \in \Sigma \cup \{\$, \}, X_1, X_2, \ldots, X_j \in \{left\, right\, stationary\} \).

0 \leq j \leq c. If \( j \leq 2 \), then \( \delta'(q_0, w, a, n) = \delta(q_0, w, a, n) \). Otherwise, \( \delta' \) has the following corresponding rules:

\[
\delta(q_0, w, a, n) = \{(q_1, w_1, X_1), (p_1, w_0, stationary)\}
\]

\[
\delta'(p_1, w_0, a, n) = \{(q_2, w_2, X_2), (p_2, w_0, stationary)\}
\]

\[
\delta'(p_{j-2}, w, a, n) = \{(q_{j-1}, w_{j-1}, X_{j-1}), (q_j, w, stationary)\}. \]

The new state set \( Q' \) includes \( Q \) and all the new states \( p_1, p_2, \ldots, p_{j-2} \) that are used in the
decomposition of the steps with three or more choices into steps with two choices.

It is clear that each move of M can be simulated by $M'$ with at most $c-1$ moves. So $(c-1)t$ steps of $M'$ are enough to simulate $t$ steps of $M$. □

Definition: Subtree Broadcast ITA (SBITA) — An SBITA $S$ is the same as an ITA except as specified in the following. Some of the machines in $S$ are designated to be control units. If $S(\alpha)$ is a control unit, then for every descendant $S(\beta)$ of $S(\alpha)$, the next state of $S(\beta)$ depends on the current state of $S(\alpha)$ as well as on the current states of $S(\beta)$ and its parent and its children. At all times, for every machine $S(\beta)$, at most one ancestor of $S(\beta)$ is a control unit. During the computation a control unit $S(\alpha)$ ceases to function as a control unit when it designates its children $S(\alpha0)$ and $S(\alpha1)$ to become control units. Initially, $S(\lambda)$ is the only control unit.

Lemma 3: Every $t$ steps of an SBITA can be simulated by $2t$ steps of an ITA.

Proof: Let $S$ be an SBITA. We design an ITA $R$ that simulates $S$.

$R(\beta)$ is designated as a control unit when the state of $R(\beta)$ is in a special set of states called the broadcast set. The broadcast set contains a particular state $q_{\text{pass}}$, and $R(\beta)$ enters state $q_{\text{pass}}$ on the step before it will pass its control unit capabilities to its descendants. If $R(\beta)$ is in state $q_{\text{pass}}$ at any time $t$, then at time $t+1$, $R(\beta)$ will be in some state not in the broadcast set. At any time $t$, the state of $R(\beta)$ can be in the broadcast set only if one of the following is true:

1. $\beta=\lambda$ and $t=0$.
2. at time $t-1$, the state of $R(\beta)$ was in the broadcast set, or
3. at time $t-1$, the parent of $R(\beta)$ was in state $q_{\text{pass}}$.

The remainder of the proof is taken directly from Seiferas (1977b), and is modified to apply to deterministic ITAs rather than nondeterministic dTAs. Informally, a machine in $R$
simulating a control unit in S performs a broadcast by propagating the values of its states to its descendants. The simulation of each descendant will lag by time equal to its distance from the control unit, so communication between a control unit and any descendant will take twice as long in R as in S.

Let $Q'$ denote the set of states of each machine in S. Let $\delta'_0$, $\delta'_1$, and $\delta'_2$ denote the transition functions of S.

Let Q denote the set of states in R, where $Q = Q' \times Q' \times Q'$, where $q = (q_0, q_1, q_2)$. When $q \in Q$, denote the first component as $\text{phase}(q)$, the second component as $\text{control}(q)$, the third component as $\text{prev}(q)$, and the fourth component as $\text{current}(q)$. $\text{Control}(q)$ propagates the states of the control unit to its descendants; $\text{prev}(q)$ holds the previous state of a machine for reference by its children; $\text{current}(q)$ holds the current state in the simulation of the corresponding machine in S.

Define for $R(\lambda)$ as control unit

\[
\delta_n(q, a, q_1, q_2) = \begin{cases} 
(0, \delta'_0(\text{control}(q), \text{current}(q), a, q_1, q_2), \text{current}(q)) & \text{if } q_n \in Q - q_1, \text{phase}(q) = 1. \\
(1, \text{control}(q_1), \text{prev}(q), \text{current}(q)) & \text{if } q_n \in Q - q_1, \text{phase}(q) = 0, r(q_1, a, q_2, q) = \text{false}. \\
(1, \text{phase}(q_1), \text{phase}(q_1), \text{phase}(q)) & \text{if } q_n = q_1, r(q_1, a, q_1, q) = \text{false}. \text{ or } \emptyset \\
\emptyset & \text{otherwise.}
\end{cases}
\]

where $q' = \begin{cases} 
\text{phase}(q) & \text{if } q_n = q_1 \text{ or } \\text{current}(q) & \text{otherwise.}
\end{cases}$

Define for $R(\beta)$, $\beta = \lambda$, where $R(\beta)$ is a control unit.
\[ \delta_i(q, q_1, q_2, q_3) = \begin{cases} 
\text{(0, } 0', (\text{control }(q_1), \text{current }(q_1), q_1', q_2', q_3'), \text{current } (q_2)) & \text{if } q_1, q_2, q_3 \in Q \land \text{phase } (q_1) = 1, \\
\text{(1, control } (q_1), \text{prev } (q_1), \text{current } (q_2)) & \text{if } q_1, q_2, q_3 \in Q \land \text{phase } (q_1) = 0, \\
\text{otherwise } . \end{cases} \]

where \( q' = \begin{cases} \text{phase } (q_1) & \text{if } q_1 = q_3, \text{or} \\
\text{current } (q_2) & \text{otherwise}. \end{cases} \]

\( \delta_i \) is defined similarly.

Define for \( R(\beta), \beta = \lambda \), where \( R(\beta) \) is not a control unit.

\[ \delta_i(q, q_1, q_2, q_3) = \begin{cases} 
\text{(0, control } (q_3), \text{current } (q_3), 0', (\text{control } (q_1), \text{current } (q_1), q_1', q_2')) & \text{if } q, q_1, q_2, q_3 \in Q \land \text{phase } (q_1) = 1, \\
\text{(1, control } (q_1), \text{prev } (q_1), \text{current } (q_2)) & \text{if } q, q_1, q_2, q_3 \in Q \land \text{phase } (q_1) = 0, \\
\text{otherwise } . \end{cases} \]

(Note: This transition propagates control information.)

Informally, the first three cases in each definition above are "simulate a transition," "wait your turn to simulate a transition," and "set up to start simulation," respectively.

This completes the definition of ITA R that simulates SBITA S.
A configuration of \( R \) is a pair \((C,w)\); a configuration of \( S \) is a pair \((C',w')\).

The sequence \( \alpha=[(C_{0,0},w_{0,0}),(C_{1,1},w_{1,1}),(C_{2,2},w_{2,2})...] \) of configurations of \( R \) is **skewed** if all of the following hold for all nonnegative integers \( j, \beta \in \{0,1\}^* \):

\[
\begin{align*}
w_{2j} &= w_{2j+1}, \\
\text{if } j \leq |\beta| &\text{, then } C_j(\beta) &= \text{control}(C_{1\beta 1}, (\beta)) = \text{current}(C_{1\beta 1}, (\lambda)). \\
\text{if } j > |\beta| &\text{, then } C_j(\beta) \in Q. \\
\text{phase}(C_j(\beta)) &= \begin{cases} 1 & \text{for } j-|\beta| \text{ odd} \\
0 & \text{for } j-|\beta| \text{ even} \end{cases}. \\
\text{prev}(C_{1\beta 1}, (\beta)) &= \text{current}(C_{1\beta 2}, (\beta)) \\
&= \text{prev}(C_{1\beta 1}, (\beta)) \\
&= \text{prev}(C_{1\beta 1}, (\beta)) \\
&= \text{phase}(C_n(\beta)). \text{ and} \\
\text{current}(C_{1\beta 2}, (\beta)) &= \text{current}(C_{1\beta 2}, (\beta)) \\
&= \text{prev}(C_{1\beta 2}, (\beta)) \\
&= \text{prev}(C_{1\beta 2}, (\beta)).
\end{align*}
\]

For such a skewed sequence \( \alpha \), define

\[
\text{skew}^{-1}(\alpha) = [(C'_{0,0},w'_{0,0}),(C'_{1,1},w'_{1,1}),(C'_{2,2},w'_{2,2})...]
\]

if the following hold for all nonnegative integers \( j, \beta \in \{0,1\}^* \), \( q = C_{1\beta 1}, (\beta) \):

\[
w_{2j} = w_{2j+1}, \text{ and}
\]

\[
\begin{align*}
C'_{j,1}(\beta) &= \begin{cases} \text{phase}(q) & \text{if } q = q \text{ or} \\
\text{current}(q) & \text{otherwise}. \end{cases}
\end{align*}
\]

The definition of a skewed sequence does not restrict the choice of values \( \text{current}(C_{1\beta 2}, (\beta)) \) for \( j > 0 \), so every sequence \( \text{current}(C_{1\beta 2}, (\beta)) \) of configurations of \( S \) where all machines are initially quiescent is equal to \( \text{skew}^{-1}(\alpha) \) for some skewed sequence \( \alpha \). We show below that \( \alpha \) is a computation by \( R \) from \( (C_{\beta 0},w_{\beta 0}) \) and onl

if \( \text{skew}^{-1}(\alpha) \) is a computation by \( S \) from \( (C_{\beta 0},w_{\beta 0}) \). It will suffice to map state \( q \in Q \) to state
phase \(q \in Q'\) if \(q = q_0\) or to state current \((q) \in Q'\) otherwise.

Let \(a\) be a skewed sequence with
\[
\alpha = [(C_{0}, w_0), (C_{1}, w_0), (C_{2}, w_1), (C_{3}, w_1), \ldots ,]
\]
and
\[
	ext{skew}^{-1}(\alpha) = [(C'_{0}, w_0), (C'_{1}, w_1), (C'_{2}, w_2), \ldots ,].
\]

To prove that \(a\) is a computation by \(R\) if and only if \(\text{skew}^{-1}(\alpha)\) is a computation by \(S\), it suffices to verify for all nonnegative integers \(j\), \(\beta \in \{0, 1\}^* \cup \{\lambda\}, y = 1\beta y_1 + 2j + 2\), that \(C_{y+1}(\beta)\) results from the transition from \(L.C(\beta, y)\) and that \(C_{1\beta y_1 + 2j + 2}(\beta)\) results from the transition from \(L.C(\beta, 1\beta y_1 + 2j + 1)\) if and only if \(C'_{y+1}(\beta)\) results from the transition from \(L.C(\beta, y)\).

Lemma 4: Every \(t\) steps of a one-head, one-tape TM can be simulated by a two-stack machine in up to \(5t\) steps.

Proof: This lemma is proved in Hopcroft and Ullman (1979).

Call the stacks of the two-stack machine stack \(A\) and stack \(B\). Let \(tp(1), \ldots , tp(j)\) denote the nonblank contents of the TM tape. The two-stack machine represents a TM configuration by holding \(tp(1), \ldots , tp(i)\) in stack \(A\) with \(tp(i)\) at the top and \(tp(i+1), \ldots , tp(j)\) in stack \(B\) with \(tp(i+1)\) at the top, where \(tp(i)\) is the tape element that the head of the TM is reading.

In five steps, the two-stack machine can simulate a TM step in which the head moves to

1. Pop \(tp(i)\) from stack \(A\).
2. Change the symbol contained in \(tp(i)\) to the symbol to be written in this step.
3. Push \(tp(i)\) back onto stack \(A\).
4. Pop \(tp(i+1)\) from stack \(B\).
5. Push \(tp(i+1)\) onto stack \(A\).
In three steps, the two-stack machine can simulate a TM step in which the head moves to the left:

1. Pop tp(i) from stack A.
2. Change the symbol contained in tp(i) to the symbol to be written in this TM step.
3. Push tp(i) onto stack B.

In three steps, the two-stack machine can simulate a TM step in which the head remains on the same tape element:

1. Pop tp(i) from stack A.
2. Change the symbol contained in tp(i) to the symbol to be written in this TM step.
3. Push tp(i) back onto stack A. ∎

Theorem 5: For all $T(n)$, every language recognized in time $T(n)$ by a $k$-tape ATM can be recognized by an ITA in time $O(T(n))$.

Proof: Let $AM$ be a one-tape ATM with at most two choices at each step. By Lemmas 1, 2, and 3, to prove the theorem it suffices to show that an SBITA $S$ can simulate $AM$ in time $O(T(n))$.

It is clear that a one-head, one-tape TM can directly implement a branch of the computation tree of $AM$. By Lemma 4, a two-stack machine can simulate the actions of a one-head, one-tape TM.

$S$ can implement the stacks in the manner described below (modeled after the implementation in Culik and Yu (1984)). For all $β$ and all $z>0$, all descendants of $S(β)$ at distance $z$ from $S(β)$ are in the same state and contain the same stack elements. Let $Δ(β)$ denote the portion of $S$ consisting of $S(β)$ and all of its descendants. $Δ(β)$ implements a pair of stacks, stack $A$ and stack $B$, each of which operates separately according to the rules below. Every machine in $Δ(β)$ is a finite state machine and can store one stack element of each stack. $S(β)$ is a control unit and holds the state of the two-stack machine and the
elements at the top of each stack. To push stack A, $S(\beta)$ broadcasts to each of its descendants $S(\alpha)$ to send the element of stack A contained in $S(\alpha)$ to both children of $S(\alpha)$, and $S(\beta)$ stores the element to be pushed. To pop stack A, $S(\beta)$ broadcasts to each of its descendants $S(\alpha)$ to send the element of stack contained in $S(\alpha)$ to the parent of $S(\alpha)$.

At the beginning of the simulation, by definition of an SBITA, $S(\lambda)$ is the only control unit. $\Delta(\lambda)$ is functioning as a pair of stacks, A and B, to store the input string $w$. $S(\lambda)$ reads $w$ and stores $w$ in stack A. $S(\lambda)$ next pops $w$ from stack A and pushes $w$ onto stack B. At this point, $w$ is stored in stack B with the first symbol of $w$ at the top of stack B. In addition, $\Delta(\lambda)$ is functioning as a pair of stacks, C and D, to simulate the actions in a branch of the computation tree of AM. This simulation continues as long as each step in the computation tree of AM has only one choice. When AM first takes a step with two choices, control unit $S(\lambda)$ broadcasts to its descendants to push all stacks one level down, and $S(\lambda)$ passes the control unit capability to $S(0)$ and $S(1)$. Then $\Delta(0)$ simulates one of the choices, and $\Delta(1)$ simulates the other choice. $S(\lambda)$ no longer acts as a control unit, but enters existential state $q_e$ if AM is in an existential state at this point or enters universal state $q_u$ if AM is in a universal state at this point. Whenever two choices are present at a node of the binary computation tree, $S$ will proceed to simulate each of the choices in the manner described above except the steps will be for $\Delta(\beta)$ instead of for $\Delta(\lambda)$. $S$ simulates each branch of the binary computation tree in parallel with the other branches.

When a branch finishes a computation, the unique control unit for that branch sends the answer (acceptance/rejection) up to its parent. $S(\beta)$, $\beta \in \{0,1\}^*$. If $S(\beta)$ is in state $q_a$, it will enter state $q_{acc}$ indicating acceptance, or enter state $q_{acc}^r$ indicating rejection, based on the table in Figure 4.1; if $S(\beta)$ is in state $q_a$, it will enter state $q_{acc}$ or state $q_{acc}^r$ based on the table in Figure 4.2.

To determine the time required for $S$ to simulate AM, look at the various components of the simulation. The reading of input string $w$ and positioning of $w$ in stack B by $S(\lambda)$
Figure 4.1 - Table of state transitions from state $q_c$.

$q_{other}$ represents any state other than $q_{acc}$ or $q_{ref}$.
Figure 4.2 - Table of state transitions from state $q_a$. 

<table>
<thead>
<tr>
<th>state of $R(\beta_1)$</th>
<th>$q_{acc}$</th>
<th>$q_{other}$</th>
<th>$q_{ref}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>state of $R(\beta_0)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_{acc}$</td>
<td>$q_{acc}$</td>
<td>$q_a$</td>
<td>$q_{ref}$</td>
</tr>
<tr>
<td>$q_{other}$</td>
<td>$q_a$</td>
<td>$q_a$</td>
<td>$q_{ref}$</td>
</tr>
<tr>
<td>$q_{ref}$</td>
<td>$q_{ref}$</td>
<td>$q_{ref}$</td>
<td>$q_{ref}$</td>
</tr>
</tbody>
</table>
requires time 2n. By Lemma 4, the two-stack simulation of each branch requires time $O(T(n))$. The concurrent SBITA simulation of two-stack machines requires time $T(n)$. The passing of the result up to the root requires time $T(n)$. Therefore the overall simulation of AM by $S$ requires time $O(T(n))$. □

Next is the ATM simulation of the dIA.

Theorem 6: For all $T(n)$, every language recognized in time $T(n)$ by a dIA can be recognized in time $O((T(n))^d)$ by an ATM.

Proof: Let $M$ be a nondeterministic dIA of time complexity $T(n)$. We design an ATM $AM$ that will simulate the operation of $M$. (Note: $AM$ will existentially guess $T(n)$.)

The local configuration of $M(\mathbf{x})$ at time $t$, denoted $LC(\mathbf{x},t)$, is the $(2d+1)$-tuple containing the state of $M(\mathbf{x})$ at time $t$ and the states of the 2d neighbors of $M(\mathbf{x})$ at time $t$.

The computation of $M$ on the input can be expressed by a $(d+1)$-dimensional computation array $C$ in which the value at coordinates $(\mathbf{x},t)$ is the state of $M(\mathbf{x})$ at time $t$, except for one column of $C$ that contains the input string. (See Figure 4.3.) For all $d$-dimensional subarrays $D$ of $C$ denote the endpoints of $D$ as $x_{1a}$ and $x_{1b}$ in dimension 1, $x_{2a}$ and $x_{2b}$ in dimension 2, ..., $x_{da}$ and $x_{db}$ in dimension $d$, and $t_a$ and $t_b$ in dimension $d+1$, where $x_{1a} < x_{1b}, x_{2a} < x_{2b}, ..., t_a < t_b$. For $k=1,...,d$, let $X_k$ denote $\{x_{1a} - 1, x_{1a}, ..., x_{1b} + 1\}$. Let $X_k'$ denote $\{x_{1a}', ..., x_{1b}'\}$. Let $(x_{1a}, x_{2a}', x_{3a}', ..., x_{ja}')$ denote the set of coordinates

$\{ (x_{1a}, x_{2a} - 1, x_{3a} - 1, ..., )$, $(x_{1a}, x_{2a}, x_{3a} - 1, ..., )$, $(x_{1a}, x_{2a} + 1, x_{3a} - 1, ..., )$, $(x_{1a}, x_{2a} - 1, x_{3a} ..., t_a )$, $(x_{1a}, x_{2a} + 1, x_{3a} ..., t_a )$, $(x_{1a}, x_{2a} + 1, x_{3a} + 1, ..., )$, $(x_{1a}, x_{2a} + 1, x_{3a} + 1, ..., ) \}$. Let $\partial D$ be the boundary of $D$. $\partial D$ contains a value for each coordinate vector in the union of
Figure 4.3 - Computation array C for 1-dimensional case
the sets

\[(x_{1n} -1, X_2, \ldots, X_d, X_{d+1}).\]

\[(x_{1o}, X_2, \ldots, X_d, X_{d+1}).\]

\[(x_{1n} +1, X_2, \ldots, X_d, X_{d+1}).\]

\[(x_{1b} -1, X_2, \ldots, X_d, X_{d+1}).\]

\[(x_{1o}, X_2, \ldots, X_d, X_{d+1}).\]

\[(x_{1b} +1, X_2, \ldots, X_d, X_{d+1}).\]

\[(X_1, X_{2o} -1, X_3, \ldots, X_d, X_{d+1}).\]

\[(X_1, X_{2o} \ldots, x_{d+1} +1, X_{d+1}).\]

\[(X_1, \ldots, X_d, \ldots, X_d).\]

Informally, \(\partial D\) consists of \(6d +2\) sets of values around the boundary of \(D\). The values at the corners of \(D\) are contained in multiple sets. Each set of values is contained on a separate tape. \(\partial C\) contains the input string.

(Note: For \(d=1\), \(\partial D\) is shown in Figure 4.4.)

Assume that when \(M(\emptyset)\) enters an accepting state, then on the following steps \(M(\emptyset)\) remains in the accepting state, and every other machine in \(M\) proceeds to enter the quiescent state.

Define \(valid(\partial D)\) to be a predicate that is true if and only if there is an assignment of states to the entries of \(D\) such that all state transitions of each machine in \(M\) follow from the transition rules of \(M\). Call such an assignment \(consistent\). The recursive procedure \(ASIMD\), defined below, computes the predicate \(valid\), that is, \(ASIMD\) returns "true" or "false." \(ASIMD\) uses a divide-and-conquer method according to time and according to each dimension of the machines in \(M\).

Initially, \(A.M\) reads the input string, existentially inserts null symbols in the input string for the steps when \(M\) does not read an input symbol, existentially guesses the
Figure 4.4 - δD, input to procedure ASIMD for 1-dimensional case
remainder of 8C, and calls ASIMD(8C). AM assumes that the length of every side of C is the smallest power of 2 greater than or equal to T(n). AM accepts the input string if and only if

1. ASIMD(8C) returns "true" and
2. the state of M(8) at time T(n) determined by ASIMD(8C) is an accepting state.

The recursive procedure ASIMD(8D) is defined as follows:

Input: 8D.

Input invariant: There exists an integer j such that for all i x_i - x_w = 2^j and t_h - t_w = 2^j.

Call 2^j the width of D.

Output: "true" or "false"

Step 1: If the width of D is 1, then check that for all \( \bar{Y} \) representing coordinates in D, the state of M(\( \bar{Y} \)) at time \( t_h = t_w + 1 \) correctly follows from \( LC(\bar{Y} , t_w) \) according to the transition rules of M. If this condition holds, then the answer returned by this invocation of ASIMD is "true"; otherwise, the answer is "false."

Step 2: At this point the width of D is greater than 1. Let \( t_m = (t_w + t_h) / 2 \). Existentially guess the states for all machines whose coordinates are in \( (X_1, \ldots, X_d) \) at time \( t_m \). For all machines whose state at time \( t_m \) is in 8D, check that the state guessed in this step is equal to the state in 8D. If the states are not equal, then return "false."

Step 3: For each \( i, 1 \leq i \leq d \), let \( x_m = (x_w + x_h) / 2 \). Existentially guess LC(\( \bar{Y} , t \)) for all \( \bar{Y} \in (X_1, X_2, \ldots, x_m, \ldots, X_d) \) and all \( t_g \leq t \leq t_h \). Check that the states in LC(\( \bar{Y} , t_g \)), LC(\( \bar{Y} , t_h \)), and LC(\( \bar{Y} , t_m \)) are equal to the corresponding states in 8D and to the corresponding states guessed in Step 2. If any of the corresponding states are not equal, then return "false." (See Figure 4.5.)

Step 4: Universally choose one of the \( 2^{d-1} \) subarrays of D created from performing divide-and-conquer in each of the \( (d+1) \) dimensions of D.
Figure 4.5 - Values known in computation array D after Step 3 of ASIMD
Step 5: Let \( S \) be the subarray chosen in Step 4. Call \( ASIMD(\Delta S) \).

Step 6: If all universal choices in Step 5 return "true," then return "true"; otherwise, return "false."

Now we prove that \( AM \) accepts if and only if \( M \) accepts.

First we show that \( ASIMD(\Delta C) \) returns "true" if and only if \( valid(\Delta C) \) is true. Let \( 2' \) be the width of \( \Delta D \), the input to \( ASIMD \). By induction on \( j \), we show that \( ASIMD(\Delta D) \) returns "true" if and only if \( valid(\Delta D) \) is true.

Basis \((j=1)\): Step 1 of \( ASIMD \) confirms that each state at time \( t_h \) follows from the states at \( t_h \), according to the transition rules of \( M \). Steps 2 and 3 confirm that state information on separate tapes for the same machine at the same time is equal. Therefore, \( valid(\Delta D) \) is true.

Conversely, if \( valid(\Delta D) \) is true, then the assignment of states to entries of \( D \subseteq \Delta D \) is consistent. Therefore, Step 1 of \( ASIMD \) confirms that each state at time \( t_h \) follows from the states at \( t_h \), according to the transition rules of \( M \), and Steps 2 and 3 confirm that state information on separate tapes for the same machine at the same time is equal. Therefore, \( ASIMD(\Delta D) \) returns "true."

Inductive hypothesis: \( ASIMD(\Delta D) \) returns "true" if and only if \( valid(\Delta D) \) is true.

Induction \((j>1)\): Assume \( ASIMD(\Delta D) \) returns "true." Let \( E, F, \ldots, G \) be the subarrays of \( D \) from Step 4 of \( ASIMD \). It follows that the calls to \( ASIMD(\Delta E), ASIMD(\Delta F), \ldots, ASIMD(\Delta G) \) returned "true." By the inductive hypothesis, \( valid(\Delta E), valid(\Delta F), \ldots, valid(\Delta G) \) are all true. Since array \( D \) is completely covered by the overlapping subarrays \( E, F, \ldots, G \), and the states assigned to overlapping locations of \( E, F, \ldots, G \) are the same since they are passed to \( ASIMD(\Delta E), ASIMD(\Delta F), \ldots, ASIMD(\Delta G) \) from \( ASIMD(\Delta D) \), and there is a consistent assignment of states to the entries of \( E, F, \ldots, G \), then there is a consistent assignment of states to the entries of \( D \). Therefore, \( valid(\Delta D) \) is true.
Conversely, assume \( \text{valid}(\mathcal{D}) \) is true. It follows that \( \text{valid}(\mathcal{E}), \text{valid}(\mathcal{F}), \ldots \)
\( \text{valid}(\mathcal{G}) \) are all true. By the inductive hypothesis, \( \text{ASIMD}(\mathcal{E}), \text{ASIMD}(\mathcal{F}), \ldots \)
\( \text{ASIMD}(\mathcal{G}) \) all return "true." Therefore, by Step 6 of ASIMD, \( \text{ASIMD}(\mathcal{D}) \) returns "true."

Now we show that if AM accepts, then M accepts. If AM accepts input string \( w \), then \( \text{ASIMD}(\mathcal{C}) \) returns "true," and the state of \( M(\mathcal{Q}) \) at time \( T(n) \) determined by \( \text{ASIMD}(\mathcal{C}) \) is an accepting state. Since \( \text{ASIMD}(\mathcal{C}) \) returns "true," \( \text{valid}(\mathcal{C}) \) is true. Because
\( \text{valid}(\mathcal{C}) \) is true and the state of \( M(\mathcal{Q}) \) at time \( T(n) \) determined by \( \text{ASIMD}(\mathcal{C}) \) is an accepting state, an assignment of states to all machines in M for all \( t \), \( 0 \leq t \leq T(n) \), that ends with \( M(\mathcal{Q}) \) in an accepting state exists, such that the states of all machines at time \( t \) result from the states of all machines at time \( t-1 \) according to the transition rules of M and input string \( w \). This implies that M accepts \( w \).

Next we show that if M accepts, then AM accepts. If M accepts input string \( w \), then there exists an assignment of states to all machines in M for all \( t \), \( 0 \leq t \leq T(n) \), such that the states of all machines in M at time \( t \) follow from the states of all machines at time \( t-1 \) according to the transition rules of M and the input string \( w \), and such that all machines are initially quiescent, and such that \( M(\mathcal{Q}) \) is in an accepting state at time \( T(n) \). As a result, \( \text{valid}(\mathcal{C}) \) is true; hence, \( \text{ASIMD}(\mathcal{C}) \) returns "true." Because \( \text{ASIMD}(\mathcal{C}) \) returns "true," and \( M(\mathcal{Q}) \) in an accepting state at time \( T(n) \), AM accepts \( w \).

Now we show that the time required for this simulation is \( O((T(n))^{d}) \). Let D denote
a \((d+1)\)-dimensional subarray of length \( k \) in each dimension. Let \( T_{AM}(k) \) denote the time complexity of the simulation. \( \text{ASIMD}(\mathcal{D}) \) is performed on computation array \( D \) which has sides of length \( k \). Then AM selects one of the subarrays \( S \) of \( D \) and calls \( \text{ASIMD}(\mathcal{S}) \). The lengths of the sides of \( S \) are \( k/2 \), so \( \text{ASIMD}(\mathcal{S}) \) requires time \( T_{AM}(k/2) \). The time to perform Steps 2 and 3 of \( \text{ASIMD}(\mathcal{D}) \) is \( O(k^d) \). The time complexity of \( \text{ASIMD}(\mathcal{D}) \) is
\[
T_{AM}(k) = T_{AM}(k/2) + O(k^d) = O(k^d).
\]
The space required for this simulation is \( O(k^d) \). In particular, the simulation of M by AM
takes time $T_{AM}(T(n)) = O((T(n))^d)$ and space $O((T(n))^d)$. □

The same techniques can be used to simulate an NdIA on an ATM in the same time.

**Corollary:** For all $T(n)$, every language recognized in time $T(n)$ by an NdIA can be recognized in time $O((T(n))^d)$ by an ATM.
Chapter 5

THE ATM SIMULATION OF THE ITA

This chapter contains a simulation of the ITA by the ATM and a proof of the correctness of the simulation. It also outlines a simulation of an ATM by a dIA.

Theorem 7: For all \( T(n) \), every language recognized in time \( T(n) \) by an ITA can be recognized in time \( O((T(n))^2) \) by an ATM.

Proof: Let \( R \) be an ITA of time complexity \( T(n) \). We design an ATM \( AM \) with four worktapes that will simulate the operation of \( R \). (Note: \( AM \) will existentially guess \( T(n) \).)

For \( \beta \in \{0,1\}^* \) and \( X \in \{0,1\} \), the local configuration of \( R(\beta X) \) at time \( t \), denoted \( LC(\beta X,t) \), is the quadruple \((q, (t), q_i(t), q_f(t), q_p(t))\) where

- \( q(t) \) is the state of \( R(\beta X) \) at time \( t \).
- \( q_i(t) \) is the state of \( R(\beta X0) \) at time \( t \).
- \( q_f(t) \) is the state of \( R(\beta X1) \) at time \( t \).
- \( q_p(t) \) is the state of \( R(\beta) \) at time \( t \) if \( \beta X = \lambda \) or \( q_p(t) \) is the input symbol at time \( t \) if \( \beta X \neq \lambda \).

Define \( |\lambda| = 0 \).

Let \( \Delta(\beta) \) denote the portion of \( R \) that comprises \( R(\beta) \) and all descendants of \( R(\beta) \). Let \( \sigma_3 \) denote the sequence of states \( q(t) \) of \( R(\beta) \) for \( 0 \leq t \leq T(n) \). Define \( \text{valid}(\sigma_{3X}, \sigma_3, \beta X) \) to be a predicate that is true if and only if there is an assignment of states to \( \Delta(\beta X) \) for every time \( t, 0 \leq t \leq T(n) \), such that for every \( i, 0 \leq i \leq T(n) \), the states of every machine in \( \Delta(\beta X) \) at time \( i \) follows from the states of every machine in \( \Delta(\beta X) \) at time \( i - 1 \) according to the transition rules of \( R, \sigma_{3X} \), and \( \sigma_3 \). Call such an assignment of states consistent. The recursive procedure \( \text{ASIMT} \) defined below computes the predicate \( \text{valid} \), that is, \( \text{ASIMT} \) returns "true" or "false."
Initially, AM writes input string \( w \) onto tape 4, existentially inserting null symbols for the steps when \( M \) does not read an input symbol. AM then existentially guesses the states for \( R(\lambda) \) at each time step from 0 to \( T(n) \), then calls \( ASIMT(\sigma_\lambda, w, \lambda) \). AM accepts if and only if

1. the output of \( ASIMT(\sigma_\lambda, w, \lambda) \) is "true," and
2. the state assigned to \( R(\lambda) \) at time \( T(n) \) by \( ASIMT(\sigma_\lambda, w, \lambda) \) is an accepting state.

The recursive procedure \( ASIMT(\sigma_{\beta X}, \sigma_{\beta X}, \beta X) \) is defined below.

Inputs: \( \sigma_{\beta X} \) (the sequence of states \( q_i(t) \)) on tape 1, \( \sigma_{\beta} \) (the sequence of states \( q_p(t) \)) on tape 4, and \( \beta X \).

Output: "true" or "false"

**Step 1:** If \( |\beta X| = T(n) \), then return "true" if all states of \( q_i(t) \) are quiescent; if not all states are quiescent, then return "false." If \( |\beta X| < T(n) \), then perform the following. Existentially guess \( \sigma_{\beta X} \) on tape 2. Existentially guess \( \sigma_{\beta X1} \) on tape 3. (See Figure 5.1.) Verify for each \( t, 0 \leq t \leq T(n) \), that \( q_i(t) \) follows from \( LC(\beta, t - 1) \), defined on tapes 1-4, according to the transition rules of \( R \). If any \( q_i(t) \) does not follow, then return "false." Verify that the symbols generated for \( q_p(t) \) are equal to the symbols already on tape 4. If they are not, then return "false." If no consistent sequence of guesses is possible, then return "false."

**Step 2:** Copy the contents of tape 1 onto tape 4.

**Step 3:** Universally choose either \( R(\beta X 0) \) or \( R(\beta X 1) \). If \( R(\beta X 0) \) is chosen, then copy the contents of tape 2 onto tape 1. then call \( ASIMT(\sigma_{\beta X 0}, \sigma_{\beta X 0}, \beta X 0) \). If \( R(\beta X 1) \) is chosen, then copy the contents of tape 3 onto tape 1. then call \( ASIMT(\sigma_{\beta X 1}, \sigma_{\beta X 1}, \beta X 1) \).

**Step 4:** If both universal choices in Step 3 return "true," then return "true"; otherwise return "false."
Figure 5.1 - Contents of ATM tapes in procedure ASIMT
Now it will be shown that AM accepts if and only if R accepts.

We show by induction on \( h = T(n) - 1 \), that \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) returns "true" if and only if \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) is true.

Basis (\( h=0 \)): If \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \), \( \|\beta X\| = T(n) \), returns "true," then the sequence of local configurations is consistent with all descendants of \( \Delta(\beta X) \) according to the transition rules of R, since all machines in \( \Delta(\beta X) \) are in a quiescent state at all times and have no effect on the computation. This implies that \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) is true.

Conversely, if \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) is true, then a consistent assignment of states to \( \Delta(\beta X) \) exists. According to the transition rules of R, all states assigned to \( \Delta(\beta X) \) must be quiescent, because all machines in \( \Delta(\beta X) \) are below level \( T(n) \) in R. Therefore, Step 1 of \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) confirms that all states in \( \sigma_{\Delta X} \) are quiescent. Therefore, \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) returns "true."

Inductive hypothesis: \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) returns "true" if and only if \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) is true.

Induction (\( h>0 \)): Assume \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) returns "true." It follows that for some \( \sigma_{\Delta X} \) and \( \sigma_{\Delta X} \), the calls to \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) and \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) returned "true." By the inductive hypothesis, \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) and \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) are both true. By Step 1 of \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \), \( \sigma_{\Delta X} \) is a sequence of states \( q, (t) \) for \( \Delta(\beta X) \), such that \( q, (t) \) follows from \( LC(\beta X, -1) \) according to the transition rules of R. Therefore, a consistent assignment of states to \( \Delta(\beta X) \) exists. Therefore, \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) is true.

Conversely, assume \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) is true. It follows that both \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) and \( valid(\sigma_{\Delta X}, \sigma_{\Delta X}) \) are true. By the inductive hypothesis, both \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) and \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) return "true." Therefore, by Step 4 of \( ASIMT, ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) returns "true."

Now we show that if AM accepts, then R accepts. If AM accepts input string \( w \), then \( ASIMT(\sigma_{\Delta X}, \sigma_{\Delta X}) \) has returned "true," and the state assigned to \( R(\lambda) \) at time \( T(n) \) by
ASIMT(σ, w, λ) is an accepting state. Because ASIMT(σ, w, λ) returns "true," valid(σ, w, λ) is true. Since valid(σ, w, λ) is true, and since the state assigned to R(λ) at time T(n) is an accepting state, R accepts w.

Next we show that if R accepts, then AM accepts. If R accepts input string w, then there exists an assignment of states to all machines in R for all t, 0 ≤ t ≤ T(n), such that the states of all machines in R at time t follow from the states of all machines at time t-1 according to the transition rules of R and input string w, and such that all machines are initially quiescent, and such that R(λ) is in an accepting state at time T(n). As a result, valid(σ, w, λ) is true. Since valid(σ, w, λ) is true, ASIMT(σ, w, λ) returns "true." Because ASIMT(σ, w, λ) returns "true," and the state of R(λ) at time T(n) is an accepting state, AM accepts w.

Now we show that the time required for this simulation is O((T(n))^2). Let T_{AM}(h) denote the time complexity of ASIMT(σ_{βX}, σ_{β}, βX), where h = T(n) - 1 ≤ X. Steps 2 and 3 of ASIMT(σ_{βX}, σ_{β}, βX) require time O(T(n)). ASIMT(σ_{βX}, σ_{β}, βX) calls ASIMT(σ_{βXY}, σ_{βX}, βXY), Y ∈ {0, 1}, which has a time complexity of T_{AM}(h-1). T(0) = O(T(n)). These terms give rise to the recurrence

T_{AM}(h) = T_{AM}(h-1) + O(T(n)).

In particular, the simulation of T(n) steps of R by AM requires time

T_{AM}(T(n)) = T_{AM}(T(n)-1) + O(T(n)) = O((T(n))^2).

This simulation requires space O(T(n)).

In order to complete the cycle of simulations from ITA to dIA and from dIA to ITA, only the simulation of an ATM by a dIA remains. In order to obtain a time bound on this simulation, first consider a simulation of an NTM by a dIA. Suppose that a (deterministic) dIA operating in time O(ε^p), where p is some constant, could simulate ε_1 steps of an NTM. Seiferas (1977a) proves that a DTM operating in time O(ε^p) can simulate ε_2 steps of a dIA. Together, these two simulations would imply that a DTM could simulate an NTM in
polynomial time; hence, the computational complexity class P would equal the computational complexity class NP. The equality P = NP is widely believed to be unlikely; hence, a polynomial time simulation of an NTM by a dIA is unlikely. Therefore, a polynomial time simulation of an ATM by a dIA is unlikely.

**Proposition 8:** For all T(n), every language recognized in time T(n) by an ATM can be recognized in time \(O(2^{\sqrt{n}})\) by a dIA.

Briefly, a (deterministic) dIA can simulate an ATM in exponential time as follows. The dIA can simply compute each branch of the computation tree of the ATM in turn. Each branch corresponds to the actions of a DTM, and a dIA is able to simulate a DTM in linear time according to Seiferas (1977b).

Theorem 7 and Proposition 8 together yield an exponential time bound for a dIA simulation of an ITA. One would expect this bound because the number of finite state machines potentially involved in a computation grows polynomially with time for a dIA, but grows exponentially with time for an ITA.
Chapter 6

CONCLUSIONS AND OPEN PROBLEMS

This thesis has presented three simulations and discussed a fourth. When combined, Theorems 5 and 6 imply that an ITA can simulate a dIA in time $O(t^d)$, and Theorem 7 and Proposition 8 imply that a dIA can simulate an ITA in exponential time.

These results and the work done in obtaining them suggest several open problems.

1. Can the time bounds of Theorems 5, 6, and 7 be improved?

2. Is there a language $L$ such that some dIA recognizes $L$ in linear time, but every ITA requires superlinear time to recognize $L$? Culik and Yu (1984) pose this question, but for real-time. One candidate considered for $L$ is a string of the form

$$L = (x_1 \# x_2 \# \ldots \# x_m \# y_1 \# y_2 \# \ldots \# y_n),$$

where $x_1, x_2, \ldots, x_m$ is an unordered list of values, and $y_1, y_2, \ldots, y_n$ is a sorted list of the same values. This candidate fails because an ITA can sort in time $O(n)$ according to Browning (1979), and, though a 2IA can sort in time $O(\sqrt{n} \log n)$ according to Thompson and Kung (1977), Nassimi and Sahni (1979), and Stout (1982), the dIA must write down the output, leading to a total time requirement of $O(n)$. A second possible candidate is a language of binary strings that represent connected d-dimensional figures.

3. How much time is required for an X-tree array to simulate an ATM? An X-tree is a binary tree with additional edges connecting all nodes at the same level in the tree. An X-tree array is an iterative array of finite state machines organized into an X-tree.

4. How much time is required for an ATM to simulate an X-tree array?

5. How can an ITA with depth as a function of the length of the input string simulate an ATM or an NTM?
6. How much time and space are required for an ATM with a limit on the number of its alternations to simulate either the dIA or the ITA?

A further possible area for future research is the alternating iterative array, such as either an alternating dIA or an alternating ITA. The addition of universal choices to non-deterministic dIAs or ITAs adds a second kind of parallelism. Initial work could be done in relating such a model to other models of computation.
REFERENCES


